Genetic Algorithms
The Use of Hill Climbing and Simulated Annealing Algorithms in Finding The Minimum of Some Mathematical Functions
Homework 1

Melinte Daria - Group 2E3

October 2022

1 Abstract

This paper presents how the minimum of some benchmark mathematical functions can be generated using Hill Climbing (with the following implementations: First Improvement, Best Improvement and Worst Improvement), along with Simulated Annealing (which is hybridising the Hill Climbing First Improvement method).

From this experiment, I came with the conclusion that depending on the function, there might be better configurations for the HC and SA algorithms, for the methods to find faster the function minimum.

2 Introduction

The following report compares 4 methods of finding the minimum values of 4 complex mathematical functions on their domain of definition.

The methods used are Hill Climbing [referred from now on as 'HC'], with the folling implementation methods:

- First Improvement [referred from now on as 'FI']
- Best Improvement [referred from now on as 'BI']
- Worst Improvement [referred from now on as 'WI']

along with Simulated Annealing [referred from now on as 'SA'], which is hybridising the HCFI method.

The 4 studied functions are:

- De Jong's 1 function
- Schwefel's function
- Rastrigin's function
- Michalewicz's function

We will compare the performance of each algorithm for any of the above function, showing the differences between accuracy of the output and execution time, based on given dimensions.

3 Methods

The code I wrote is based on the pseudocode found at the folling link. The algorithms used are Hill Climbing (First Improvement, Best Improvement and Worst Improvement) and Simulating Annealing.

Every function has the selected value represented as an array of bits. Because each function has a different domain of definition, the number of bits used for every function is different, and is calculated by the folling formula

$$bitStringLength = \lceil \log_2(10^{precision} * (h - l) \rceil$$

where:

- l is the ler bound of the function
- h is the upper bound of the function
- *precision* represents the number of decimals the response is going to have

The conversion of a bit string to a decimal number was done using the following formula:

$$x = \frac{bitStringToDecimal(x_{binary-format}) * (h - l) + l}{2^{bitStringLength} - 1}$$

where: x is a value between l and h and *bitStringToDecimal* is a function that accepts a bit string as parameter and return its integer value in base 10 format.

3.1 Hill Climbing [HC] Algorithm - Description

A hill-climbing algorithm is a local search algorithm that moves continuously upward (increasing) until the best solution is attained. This algorithm comes to an end when the peak is reached.

This algorithm has a node that comprises two parts: state and value. It begins with a non-optimal state (the hill's base) and upgrades this state until a certain precondition is met. The heuristic function is used as the basis for this precondition. The process of continuous improvement of the current state of iteration can be termed as climbing. This explains why the algorithm is termed as a hill-climbing algorithm.

A hill-climbing algorithm's objective is to attain an optimal state that is an upgrade of the existing state. When the current state is improved, the algorithm will perform further incremental changes to the improved state. This process will continue until a peak solution is achieved. The peak state cannot undergo further improvements.

- **First Improvement**
  This method is searching for a neighbour with a better value, by negating a random bit from the bit string.
- **Best Improvement**
  This method is searching through all the possible neighbours that can be formed from the generated bit string and is choosing the one that gives the best improvement.
- **Worst Improvement**
  This method is searching through the possible neighbours that can be formed from the generated bit string and is choosing the one that gives the smallest improvement.

3.2 Simulating Annealing [SA] Algorithm - Description

Simulated annealing is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. It is often used when the search space is discrete.

The name of the algorithm comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. Both are attributes of the material that depend on their thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy or Gibbs energy. Simulated annealing can be used for very hard computational optimization problems where exact algorithms fail; even though it usually achieves an approximate solution to the global minimum, it could be enough for many practical problems.

This notion of slow cooling implemented in the simulated annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored. Accepting worse solutions allows for a more extensive search for the global optimal solution. In general, simulated annealing algorithms work as follows. The temperature progressively decreases from an initial positive value to zero. At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and moves to it according to the temperature-dependent probabilities of selecting better or worse solutions, which during the search respectively remain at 1 (or positive) and decrease towards zero.

3.3 The Functions Used

3.3.1 De Jong's function

De Jong's function is also known as sphere model. It is continuos, convex and unimodal.

$$f(x) = \sum_{i=1}^{n}(x_i^2), x_i \in [-5.12, 5.12]$$

The global minimum:

$$f(x) = 0, x_i = 0, \forall i \in [1, n], \forall n \in \mathbb{N}$$

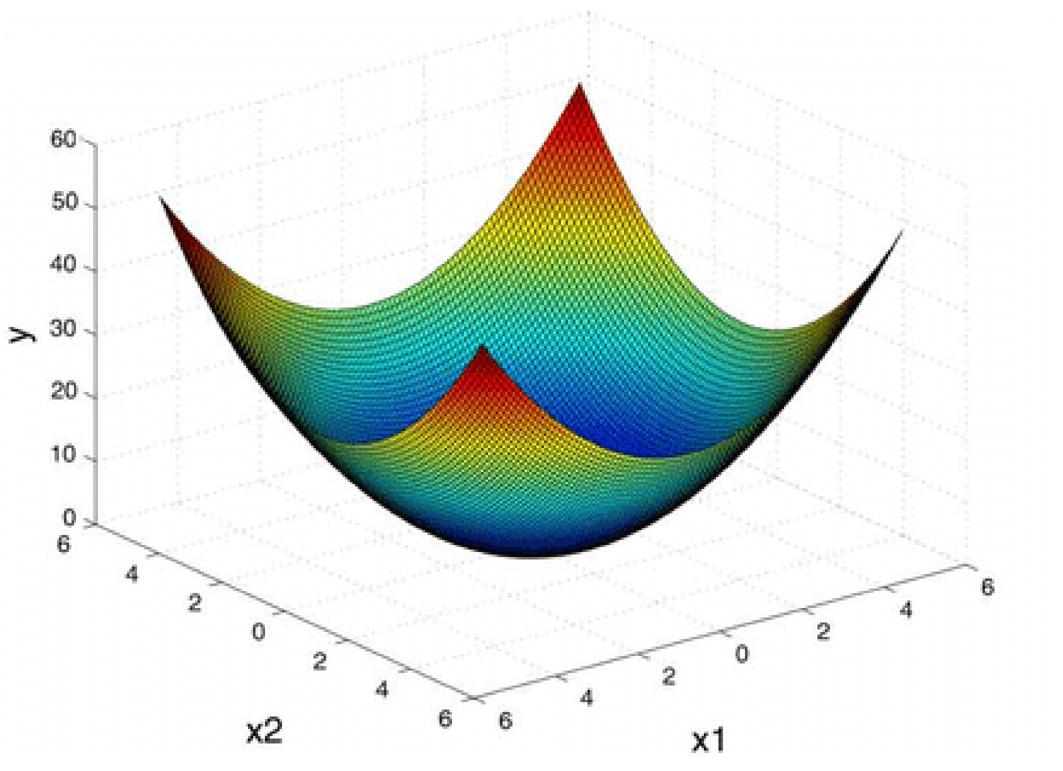

Fig. 1. De Jong's function for n = 2

3.3.2 Schwefel's function

Schwefel's function is deceptive in that the global minimum is geometrically distant, over the parameter space, from the next best local minima. Therefore, the search algorithms are potentially prone to convergence in the wrong direction.

$$f(x) = \sum_{i=1}^{n}(-x_i \sin\sqrt{|x_i|}), x_i \in [-500, 500]$$

The global minimum:

$$n = 5 : f(x) = -2094.91450$$

$$n = 10 : f(x) = -4189.82900$$

$$n = 30 : f(x) = -12569.48700$$

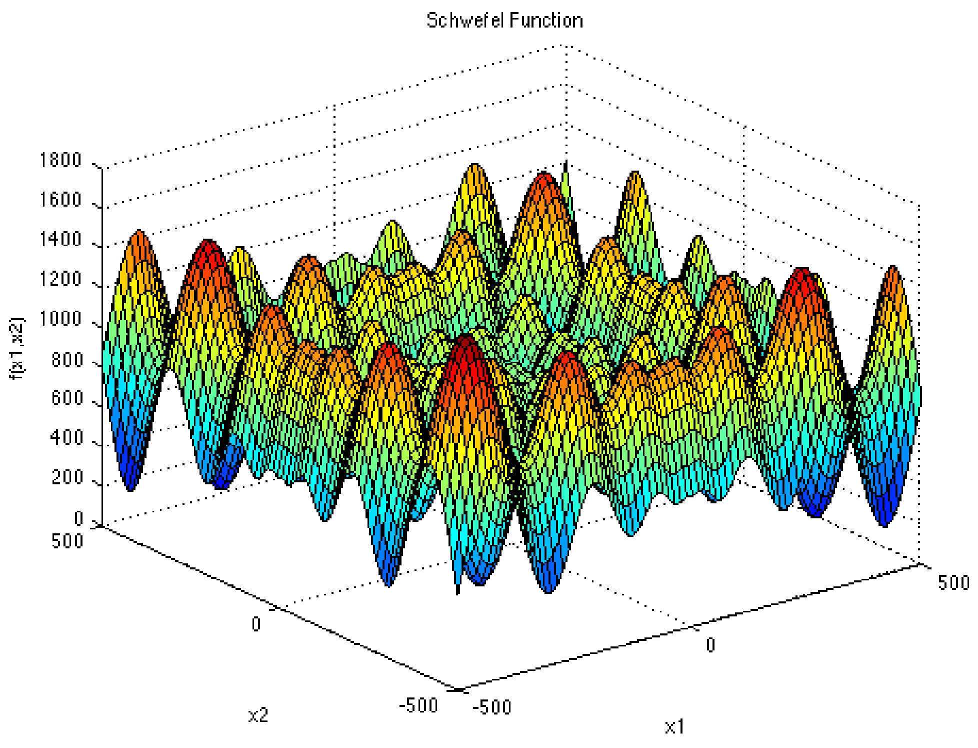

Fig. 2. Schwefel's function for n = 2

3.3.3 Rastrigin's function

Rastrigin's function is based on De Jong's first function with the addition of cosine modulation to produce many local minima. Thus, the test function is highly multimodal. However, the location of the minima are regularly distributed.

$$f(x) = 10n + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i)), x_i \in [-5.12, 5.12]$$

The global minimum:

$$f(x) = 0, x_i = 0, \forall i \in [1, n], \forall n \in \mathbb{N}$$

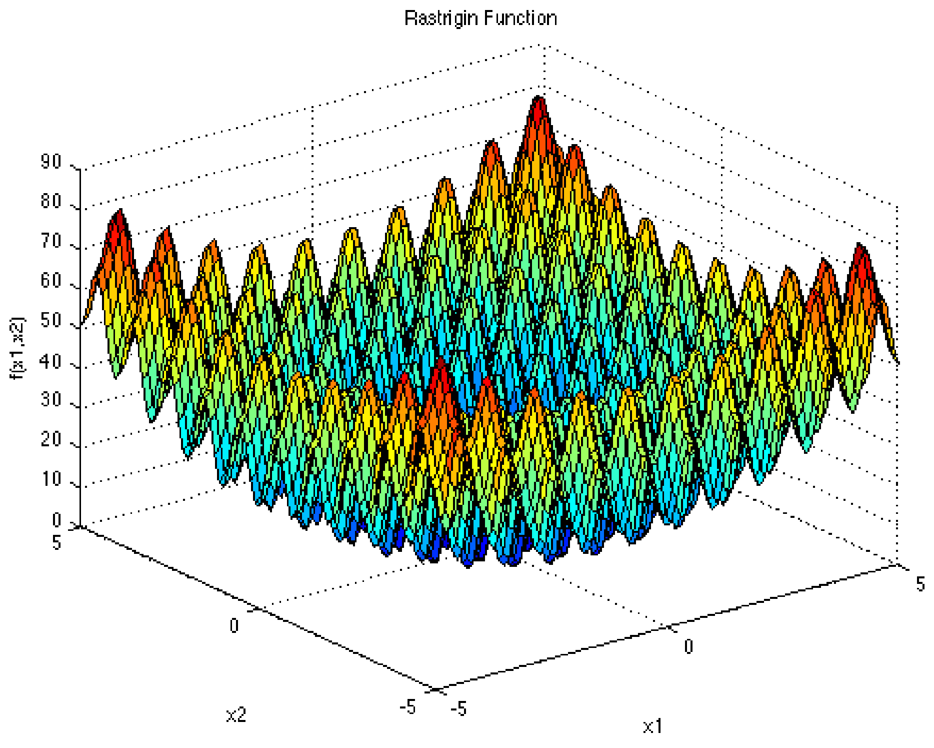

Fig. 3. Rastrigin's function for n = 2

3.3.4 Michalewicz's function

The Michalewicz function is a multi-modal test function (n! local optima). The parameter m defines the "steepness" of the valleys or edges. Larger m leads to more difficult search. For very large m the function behaves like a needle in the haystack (the function values for points in the space outside the narrow peaks give very little information on the location of the global optimum).

$$f(x) = -\sum_{i=1}^{n}(\sin(x_i)(\sin(\frac{ix_i^2}{\pi})^{2m}), x_i \in [0, \pi], m = 10$$

The global minimum:

$$n = 5 : f(x) = -4.68700$$

$$n = 10 : f(x) = -9.66000$$

$$n = 30 : f(x) = -29.63088$$

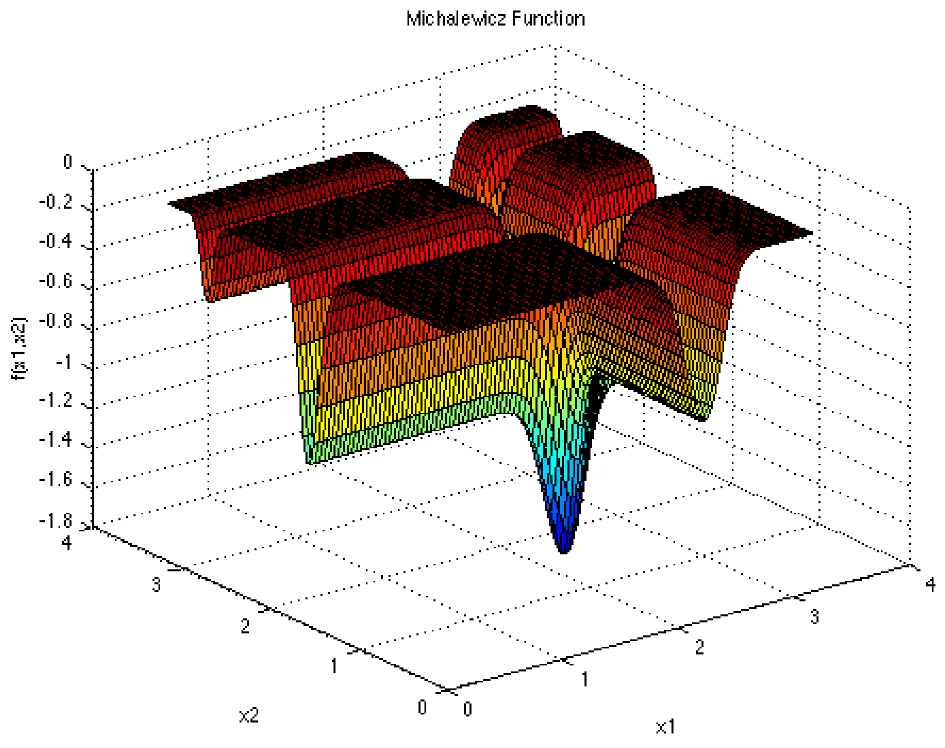

Fig. 4. Michalewicz's function for n = 2

4 Experiment

Each function has been tested on 5 and 10 dimensions, with 30 tests (samples) for each method. On 30 dimensions, I chose to test the functions with 10 samples for each method because of the long time it took for it to execute. The result is represented with a precision of 5 decimals, which seemed like a good balance between high accuracy and short execution time.

The number of iterations can be seen in the following tables. For each version of the algorithm, the number of iterations is different, based on the dimensions the functions had at the moment.

4.1 De Jong's Function

De Jong's Function	5 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	0.00000			
best	0.00000	0.00000	0.00000	0.00000
worst	0.00000	0.00000	0.00000	0.99496
median	0.00000	0.00000	0.00000	0.00000
standard deviation	0.00000	0.00000	0.00000	0.48766
IQR	0.00000	0.00000	0.00000	0.99496
median time	209.8388	93.52918	545.3906	115.9357
	10 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	0.00000			
best	0.00000	0.00000	0.00000	2.98993
worst	0.00000	0.00000	0.00000	7.44644
median	0.00000	0.00000	0.00000	5.45652
standard deviation	0.00000	0.00000	0.00000	0.89826
IQR	0.00000	0.00000	0.00000	0.939795
median time	258.6905	143.3789	311.9806	212.3537
	30 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	0.00000			
best	0.00000	0.00000	0.00000	31.51372
worst	0.00000	0.00000	0.00000	44.22246
median	0.00000	0.00000	0.00000	36.07491
standard deviation	0.00000	0.00000	0.00000	4.11809
IQR	0.00000	0.00000	0.00000	5.99669
median time	1189.208	1031.191	2061.897	434.8187

Conclusions for De Jong's function:

The De Jong's first function is one of the most straightforward functions, giving the same constant result. Its nature makes it easy of the Hill Climbing algorithm to find the minimum of the function almost immediately.

The Simulated Annealing algorithm is also giving great results, but at the cost of efficiency. The most striking thing I noticed was the improved performance of the SA algorithm on bigger dimensions.

For SA, the results from the table above were generated using a cooling number ALPHA=0,3. A bigger cooling number was of course giving more accurate results, but I found this number as the perfect balance between fast execution time and optimum results.

4.2 Schwefel's Function

Schwefel's Function	5 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	-2094.91450			
best	-2094.9144	-2094.91442	-2041.14743	-2094.91345
worst	-2094.6028	-2094.7071	-2014.16144	-2060.57399
median	-2094.80918	-2094.81076	-2014.26477	-2094.70677
standard deviation	0.07956	0.05810	13.12822	9.75479
IQR	0.10365	0.10304	26.75349	0.20695
median time	411.5498	173.6443	545.3906	118.6016
	10 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	-4189.82900			
best	-4087.01734	-4189.51565	-3920.99891	-4155.5902
worst	-3883.67884	-3967.77956	-3760.02678	-3786.68481
median	-3968.36961	-4086.55117	-3861.46654	-3939.7112
standard deviation	58.57436	52.23444	45.63767	78.43227
IQR	84.46999	66.70495	37.82786	63.19951
median time	554.1237	292.4811	1200.181	172.0283
	30 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	-12569.48700			
best	-10967.9358	-11462.39751	-11133.68864	-11003.74321
worst	-10513.09387	-11121.92563	-10461.73795	-10425.30368
median	-10748.13374	-11265.917255	-10588.36648	-10646.05907
standard deviation	168.59320	121.07910	199.94955	148.24776
IQR	294.54498	148.34054	116.78042	213.26382
median time	1233.792	2217.862	1473.663	599.7924

Conclusions for Schwefel's function:

Schwefel's function gave almost perfect accuracy on small dimensions, but as the number of dimensions increased, the accuracy of the HC and SA decreased along.

Still, the time in which SA managed to make the computations are very striking, especially on 30 dimensions, where the median time for SA is almost half as small as the one for HC.

Here, after multiple tries, I came to the conclusion that a cooling number of 0.65 works best for the function, other variations slowing the execution of the program.

4.3 Rastrigin's Function

Rastrigin's Function	5 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	0.00000			
best	0	0	0	0
worst	0.99496	0.99496	2.00002	1.00001
median	0	0	1.00001	0
standard deviation	0.46374	0.30359	0.43226	0.50541
IQR	0.99496	0	0.17685	0.99496
median time	206.1306	74.43	1237.099	683.9208
	10 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	0.00000			
best	2.23078	2.23078	5.23587	4.22575
worst	5.93825	4.70748	9.99933	7.44644
median	4.46156	3.4666	7.353795	5.58201
standard deviation	1.02384	0.65894	1.15128	0.86044
IQR	1.98486	0.99496	1.412685	0.92307
median time	226.0727	114.08	515.9031	1021.959
	30 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	0.00000			
best	36.32933	21.83114	39.39459	33.24138
worst	42.91354	35.12975	50.62801	44.76401
median	40.0004	32.006165	47.512765	39.46713
standard deviation	2.04623	3.85514	2.90894	3.20104
IQR	2.14308	3.85159	1.35426	3.67694
median time	467.7573	825.7466	5144.311	758.2673

Conclusions for Rastrigin's function:

The interesting result I got from Rastrigin's function is that HCBI performs really well on it, on smaller dimensions. On the other side, the striking part was seeing how bad the HCWI is performing on Rastrigin's function on bigger dimensions.

Another interesting this I noticed was during testing the cooling number, while running SA. While Rastrigin seems to perform well with numbers such as 0.6 or 0.7 on smaller dimensions, when the number of dimensions was increased, the slow computation could have been easily seen. I found out that for bigger dimensions, a cooling number like 0.3 works great, similar to De Jong's function. I believe this is happening because overall Rastrigin's function is a deviation of De Jong's function, so it might have "inherited" these kind of properties.

4.4 Michalewicz's Function

Michalewicz's Function	5 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	-4.68700			
best	-4.68766	-4.68766	-4.68713	-4.68766
worst	-4.68593	-4.68532	-4.66646	-4.68591
median	-4.68759	-4.68764	-4.67047	-4.68751
standard deviation	0.00043	0.00063	0.00646	0.00051
IQR	0.00013	0.00013	0.01123	0.00070
median time	105.846	62.03143	216.8206	602.69
	10 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	-9.66000			
best	-9.60625	-9.58665	-9.3071	-9.51136
worst	-9.31353	-9.27662	-8.56723	-9.14475
median	-9.408685	-9.39694	-8.77951	-9.310765
standard deviation	0.08714	0.08580	0.17144	0.09172
IQR	0.12531	0.12856	0.11405	0.14261
median time	412.259	141.4643	517.9206	607.9623
	30 dimensions			
	HCFI	HCBI	HBWI	SA[FI]
actual value	-29.63088			
best	-26.60306	-27.36084	-24.09767	-26.82681
worst	-25.73246	-26.55666	-22.61183	-25.67059
median	-26.033265	-26.978745	-23.13036	-26.09066
standard deviation	0.27988	0.26775	0.48315	0.33037
IQR	0.35630	0.401607	0.68375	0.49607
median time	763.7577	1426.031	4210.193	1276.721

Conclusions for Michalewicz's function:

I found out that Michalewicz's function performs well in all kind of conditions, giving almost exact results on every dimension that is executed upon.

Again, it seems to perform well on every runned algorithm, but especially well on HCBI.

For SA, after testing it with different cooling numbers, I noticed that the function performs really well with an ALPHA=0.25. Not only it has high performance, but the results are quite accurate too.

5 Conclusion

A first conclusion would be that every function behaves differently under different parameters, and serializing them with the same parameters might increase the computation time significantly. Relevant for this example is Rastrigin's function, where a cooling number of 0.7 was making it execute an iteration in 2146.02967 seconds (with perfect result) and after changing the cooling number to 0.3, it even had the same output (0.00000) with a computation time of 154.92141 seconds.

At the core, SA algorithm is just another form of HC algorithm, but with a twist. Sometimes, it chooses a wrong result from the neighbourhood, in order to find another local minima. This happens less and less, though, as the algorithm progresses, in time acting almost like a HCFI, if the cooling number is not chosen correctly, or even having a bigger execution time. It is needless to say, though, that SA performs better on functions with bigger number of dimensions.

6 Bibliography

- Genetic Algorithms Page - Pseudocode for HC and SA algorithms
  https://profs.info.uaic.ro/ eugennc/teaching/ga/

- Informations about the benchmark functions
  http://www.geatbx.com/docu/fcnindex-01.html

- Fig. 1.
  https://www.researchgate.net/figure/llustrates-the-2D-Dejong-Dejong-2D-and-2D-Ackley-Ackley-2D-functions-The-selected$_fig4_2$85729352

- Fig. 2.
  https://www.sfu.ca/ ssurjano/schwef.png

- Fig. 3.
  https://www.sfu.ca/ ssurjano/rastr.png

- Fig. 4.
  https://www.sfu.ca/ ssurjano/michal.png

- The site used for generating LaTeX tables
  https://www.tablesgenerator.com/latex$_tables$

- LaTeX tutorials
  https://www.overleaf.com/learn/latex/$Learn_LaTeX_in30_minutes$

- The report which holds the global minimum for Michalewicz's function, for n = 30 (Page 5, Table 3)
  https://arxiv.org/pdf/2003.09867.pdf

- Random Number Generator in C
  https://www.scaler.com/topics/random-number-generator-in-c/?fbclid=IwAR3iQmZmWwh0E-8A0HStfeBkYrvxbOdCzUpcKOX-OcV5zAQqzGQLK1ao0b4

- Hill Climbing
  https://en.wikipedia.org/wiki/$Hill_climbing$

- Simulated Annealing
  https://en.wikipedia.org/wiki/$Simulated_annealing$